TP3 - 3D Modelling

Drawing geometric primitives in OpenGL.

Modélisation et rendu, 2019-2020
2nd year, Multimedia track

Session 3

Version: v0.3.2019
(master @ e0b3cd5 2019-01-27)

Contents

1 Objective

In this TP we review the concepts that have been introduced in the last TP about the OpenGL pipeline and the scene modelling. In this TP we will introduce another aspect of the scene modelling: so far we used some built-in function to draw 3D objects, now we will start to see how we can build our own 3D object using some low level geometric primitives such as points and lines.

In Section 2 you find a brief introduction on how to draw geometric primitives in OpenGL, then in Section 3 you are asked to implement an interactive OpenGL program that will cover all the different aspects we saw in the last TP and the OpenGL geometric primitives. Finally, the last exercise in Section 4 will show with an example how to approximate a surface (a sphere) using an icosahedron and a simple subdivision algorithm.

2 Drawing primitives in OpenGL

In the last TP we saw that GLUT provides a number of nice high-level objects (cones, spheres, cubes, teapots *etc.*). All these high-level objects are ultimately composed of points (or vertices) that are connected with lines to form the wire-frame of the object. If we want to build our own object, we then have to express it in terms of more low-level primitives, such as points, lines, triangles *etc.*

OpenGL provides a number of primitives including points, lines, and polygons. Each of these objects is ultimately made up of an (ordered) collection of vertices. Vertex coordinates are specified with the OpenGL function `glVertex` (here the doc):

```
void glVertex3f(GLfloat x,  GLfloat y,  GLfloat z);
```

Vertices must be grouped together to form a collection of points (or lines, or polygons, *etc.*). The beginning and end of each group is denoted by `glBegin` and `glEnd`, and the parameter passed to `glBegin` tells OpenGL how to interpret the collection of vertices that follows (here the doc). For example, the following specifies a collection of individual points (which happen to form the corners of a box)[1]:

```
glBegin(GL_POINTS);
  glVertex3f(1,1,1);
  glVertex3f(1,2,1);
  glVertex3f(2,2,1);
  glVertex3f(2,1,1);
glEnd();
```

Since `GL_POINTS` is specified, this is drawn as four individual points. Changing `glBegin`'s parameter changes the interpretation. For example, if `GL_LINES` is passed, two lines are drawn, the first having the first two points as start and end points, respectively, the second having the other two as start and end points. You can see in Figure 1 other types of primitives that are supported by OpenGL and how the list of vertices is interpreted to generate the corresponding primitive.

Finally you can specify the current drawing color for each vertex (or primitive):

```
glBegin(GL_POINTS);
  glColor3f(1,0,0);
  glVertex3f(1,1,1);  // a red vertex
  glVertex3f(1,2,1);  // another red vertex
  glColor3f(0,0,1);
  glVertex3f(2,2,1);  // a blue vertex
  glColor3f(1,0,1);
  glVertex3f(2,1,1);  // a magenta vertex
glEnd();
```

This piece of code draws the first two points in red, the third in blue, and the last in magenta. OpenGL also provides some functions to specify the size or the width of each primitive. For example, `glLineWidth( GLfloat w);` set the current width of the line to draw. For a point you can use `glPointSize( GLfloat w);`. Again, since OpenGL is a state machine the current value for the size or the width will remain in place until it is further changed.

[1] The indentation after `glBegin` is not necessary, but improves the readability of the code.

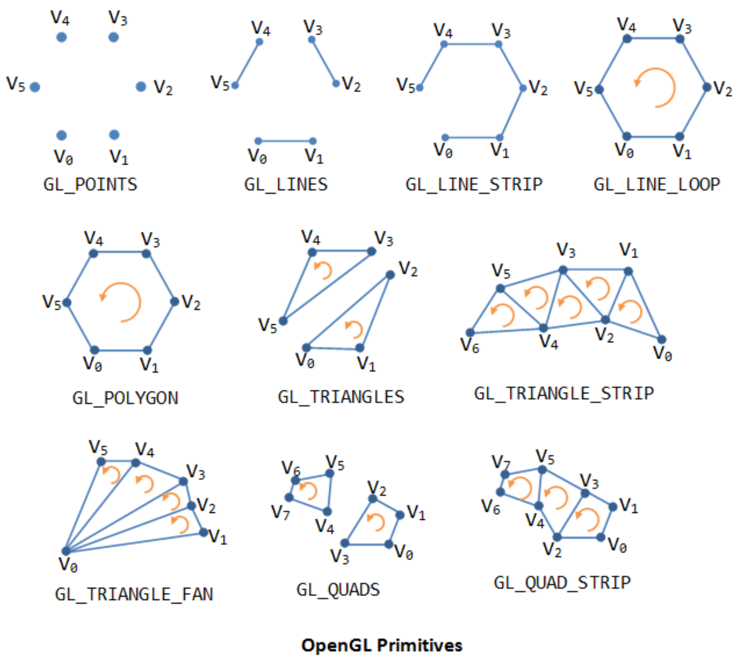

Figure 1: OpenGL geometric primitive types: the picture shows how a sequence of vertices $\{V_0, \ldots, V_n\}$ defined between `glBegin` and `glEnd` is interpreted by OpenGL, according to the type of primitive (`GL_POINTS`, ...) passed to `glBegin`.

3 Exercise 1 - The Robotic Arm

We want to implement an interactive program that allows the user to move a robotic arm. The robotic arm is composed of 3 connected joints (see Figure 2), and each of them can rotate along one axis. Each joint is sketched using a wire-frame of a parallelepiped. Moreover, to help you out with the rotation, a (local) reference system is also drawn at the base of the joint. We can assume that each joint can rotate about its x-axis (the red one in Figure 2). The user can control the movement of each joint using the keyboard. In particular

- A and Z control the movement (rotation about its x-axis) of the second joint
- E and R control the movement (rotation about its x-axis) of the third joint
- the arrow keys ←, → and ↑, ↓ control the rotation of the whole arm (or, equivalently, the first joint) about its x-axis and y-axis, respectively.

3.1 The code

In order to implement the program you can fill and complete the file `robot.c`. It comes with its makefile. The code is composed as usual by a `main` routine that initializes the OpenGL's pipeline and some other functions that you should be now familiar with:

- `init` function that initializes the camera type and its position; there is nothing you have to do here, but if you want you can change the `glLookAt` function in order to move the camera in case you want a different point of view.

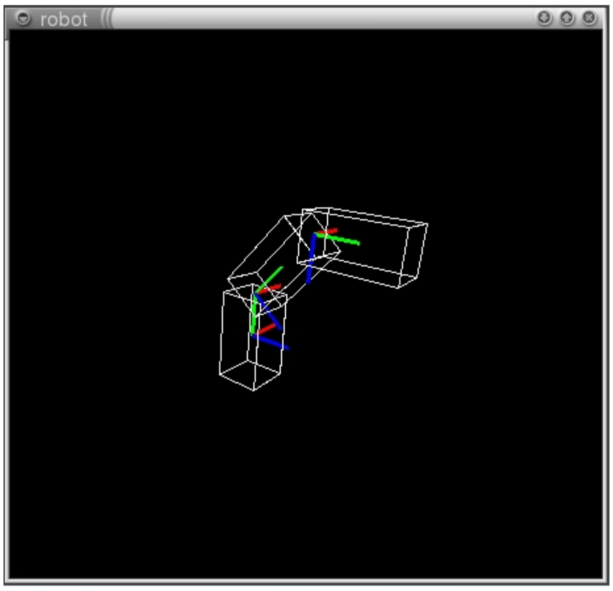

Figure 2: Example of the robotic arm.

- `reshape` callback function to set the viewport; you don't need to modify it.
- `usage` function is used to print a simple help on the screen; you don't need to modify it.
- `arrows` and `keyboard`, the callback functions to manage the user input from the keyboard; you have to complete them.
- `display`, the display callback function; you have to complete it.
- `DrawReferenceSystem`, `DrawJoint` and `DrawRobot` are the functions you have to implement to draw the robotic arm.

You also have 4 global variables, `Angle1`, `Angle2`, `RobotAngleX`, and `RobotAngleY`, which will be used to control the robot movements.

3.2 Let's implement it

The program is conceived to be highly modular so that you can implement the program incrementally and have always some visual feedback that may help you to verify and debug your code.

The whole robotic arm is composed of three joints and each joint is composed by the parallelepiped and the reference system (see Figure 3). Therefore, we will start by developing the function `DrawReferenceSystem` that draws the reference system, then the function `DrawJoint` that draws the joint as a parallelepiped and a reference system, and finally the function `DrawRobot` that draws the whole arm as a system of 3 joints. If you implement the functions correctly at each step you should have a visual feedback of what you have coded. Finally, you will implement the part of the code that manages the input from the user and allows to move the robot.

These are the main steps you can follow:

1. implement `DrawReferenceSystem`: the function should draw 3 perpendicular lines corresponding to the 3 directions x, y, z (see Figure 3). Follow the comments in the code and

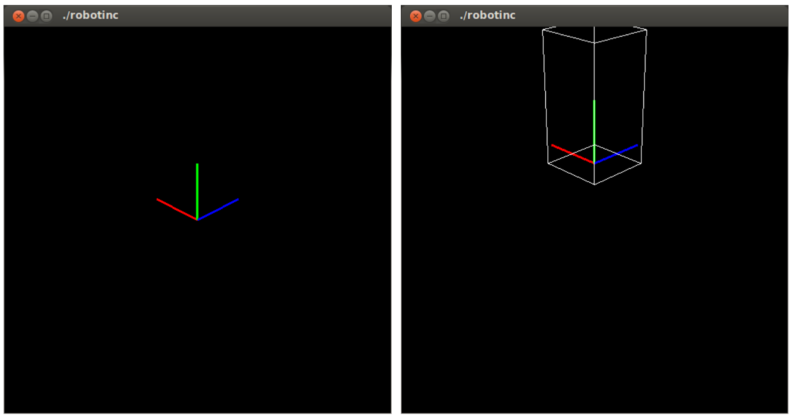

Figure 3: The visual output after you correctly finish to implement `DrawReferenceSystem` (a) and `DrawJoint` (b). Note that the views has been zoomed in.

remember that you can draw lines using the OpenGL function that has been introduced at the beginning in Section 2.

2. implement `DrawJoint`: the function should draw a parallelepiped and a reference system placed at the origin of its local reference system. In order to draw the parallelepiped you can use the function `glutWireCube` and use `glScalef` to stretch it along, *e.g.*, its y-axis. Another option is to draw the parallelepiped defining its vertices using the `GL_QUAD` primitive.

   - `glutWireCube` draws the cube so that it is centered in the origin of the reference system. Hence you have to move it in order to have the reference system drawn on its bottom face (here the doc).
   - Be careful when using `glScalef` (think in terms of the current modelview matrix. . .), you might want that it only affects the local coordinate system of the parallelepiped.

3. implement `DrawRobot`: it must draw three joints, one on top of the other (on the y-axis). You can add the rotation of each joint now or later when you implement the user control.

4. Now we have the robotic arm. Complete the `display` function.

5. Complete the `keyboard` function that updates the values of the rotation angles of each joint. A reasonable update value for the angle value is $\pm 5°$, but you are free to experiment with other values. If you didn't do before, add the relevant rotations in `DrawRobot`.

6. Complete the `arrows` function that updates the values of the rotation angles of the first joint using the arrows. If you didn't do before, add the relevant rotations in `display`.

7. Finally complete the `main` in order to register the callback for `keyboard` and `arrows`;

8. Add a pair of pincers (« pinces ») at the end of the last joint. A simple way to do it is to add two "vertical" rectangles that slide over another "horizontal" rectangle (*c.f.* Figure 4). Add two other keys to control the pincers (*e.g.* O and L), and manage the movement of the pincers so that they stop whenever they reach the limits of the opening and closing positions.

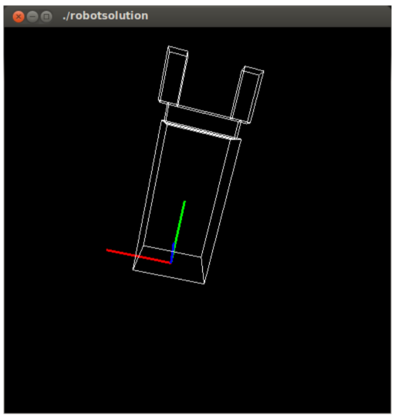

Figure 4: The last joint of the robot with a pair of pincers attached to its end.

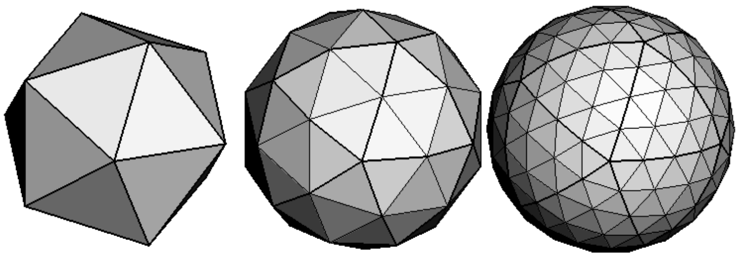

Figure 5: An example of icosahedron (on the left) and the surfaces that are generates using the recursive subdivision (middle, recursion with depth=1, right recursion with depth=2.

4 Exercise 2 - Sphere Approximation

In this exercise we want to build our code to draw a sphere by building a triangular mesh that approximate the exact surface of a sphere. When drawing surfaces, we always need to find the good trade-off between the quality and the computational speed: a mesh with many triangles gives more details and better approximates the real surface; on the other hand, the computational cost of processing and storing the mesh will be higher.

In this case, in order to approximate the sphere, we start with a regular icosahedron, which is a regular solid composed 12 vertices and 20 faces, each of which is an equilateral triangle. An icosahedron can be considered a first rough approximation for a sphere. Figure 5 shows the main idea: by subdividing the surface of the icosahedron we generate a new surface that better approximates the sphere. The algorithm is indeed quite simple:

- start with an icosahedron inscribed inside the unitary sphere
- for each triangular face:
  - find the midpoint for each edge of the face;
  - project the 3 midpoints on the unitary sphere so that we have 3 new vertices;
  - draw 4 new triangles, one connecting the 3 new vertices, the other 3 connecting two new vertices with the original vertex shared by the respective edges: *e.g.*, let v_1, v_2, v_3 be the original vertex, and v_{12}, v_{23}, v_{31} the new vertices[2], one of the 3 triangles

[2] with v_{ij} the vertex generated by the midpoint of the edge connecting v_i and v_j.

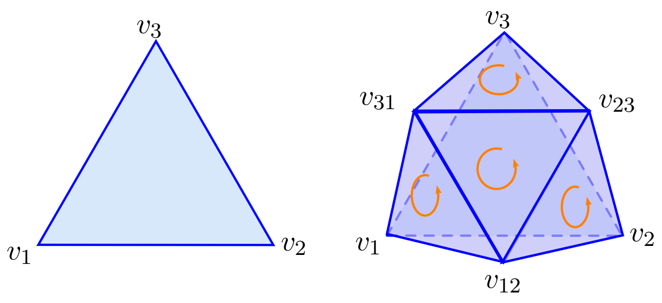

Figure 6: The original triangular face (left) and an example of the 4 new triangles generated by the subdivision algorithm (right).

will connect v_3-v_{31}-v_{23} *etc.* (*c.f.* Figure 6).

- this can be done once or recursively up to the desired approximation level.

4.1 The code

Complete the program that you find in `subdivision.c`. If you run it, nothing is drawn. On the other hand it has already the basic OpenGL structure implemented, and you can move in the scene using the ZS key combination and the arrow keys to orbit around the object. You have to implement the functions for drawing and subdividing the triangles.

In the code the vertices of the icosahedron are stored in the global variable `vertices`:

```
static float vertices[12][3] =
{
  {-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0, -Z},
  {0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
  {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
};
```

where `X` and `Z` are "magical" numbers. The triangles of the mesh are instead defined in terms of vertex indices in `vIndices`:

```
static GLuint vIndices[NUMTRIANGLES][3] =
{
  {1,4,0}, {4,9,0}, {4,5,9}, {8,5,4}, {1,8,4},
  {1,10,8}, {10,3,8}, {8,3,5}, {3,2,5}, {3,7,2},
  {3,10,7}, {10,6,7}, {6,11,7}, {6,0,11}, {6,1,0},
  {10,1,6}, {11,0,9}, {2,11,9}, {5,2,9}, {11,2,7}
};
```

where *e.g.* the first triangle is composed of `vertices[vIndices[0][0]][]`, `vertices[vIndices[0][1]][]`, and `vertices[vIndices[0][2]][]`.

4.2 Let's implement it

Here are some steps you can follow:

1. complete the function `drawtriangle()` that draws a single triangle; it takes as input the three vertices to draw; we saw in Section 2 how to draw the triangles. Instead of the usual `glVertex3f()` that takes the three coordinates of the point, you can use its vector counterpart `glVertex3fv(float *v)` that takes the vertex coordinates directly as a 3

elements vector. Finally, check also Figure 1 for the correct order (counter-clockwise) of the vertices.

2. Now you can draw the initial icosahedron: add the code to draw all the faces to `display`. Be careful with the pointers: for each vertex you need to pass to `drawtriangle()` the pointer to (or, equivalently, the address of) the first element of each vertex. Compile and see the output.

3. Hmmm... a bit disappointing, isn't it? We still don't know how to use the light to have a better rendering of the surfaces, so the surface just appear as an uniform mass of the same color. You can improve it a little bit by drawing the edges of the surface, so that the surface looks somehow like Figure 5: modify `drawtriangle`: after drawing the triangles we can draw just the edges of the triangle. Check again Figure 1 to see how this can be done. Also, you may want to use a different color for the edges, let's say black with a line width of 3.

4. Now let's build the subdivision algorithm and apply one step of the subdivision to the icosahedron. Complete the function `subdivide()` that subdivides a face into 4 new faces. You don't need to allocate the memory for the new vertices, just draw them. The function takes the 3 vertices of a face and apply the algorithm detailed above.
   For the algorithm you can implement and use the function `normalize()` to project the midpoints on the unitary sphere: it takes a vector of three elements and normalizes it. Again, respect the counter-clockwise order of the vertices when you draw the triangle

5. Once you have implemented the subdivision, replace the previous call to in `drawtriangle()` with the the call to `subdivide()`.

6. If you run the code you can see that the quality of the sphere has improved.

7. Now we want to implement the recursive version of `subdivide()` that applies the subdivision recursively for a given number of times. Complete the function `recursiveSubdivide()` that takes also the integer parameter `depth` that indicates the level of recursion. The recursion stops when `depth == 0`: if we call `recursiveSubdivide()` passing `depth = 0`, we expect to see the original icosahedron.

8. Once you have implemented the recursive version, you can bind two keys PgUp and PgDn so that they increment and decrement the value of the global integer variable `depth`. Decrement it only if it has positive values.

9. Finally you can replace the call to `subdivide()` with its recursive version passing the global variable `depth` as parameter.

10. Move the surface with the arrow keys. Try it while changing the level of approximation. What do you observe?